Side effects are not sufficient to authenticate software

Umesh Shankar[*]
UC Berkeley
ushankar@cs.berkeley.edu

Monica Chew
UC Berkeley
mmc@cs.berkeley.edu

J. D. Tygar
UC Berkeley
tygar@cs.berkeley.edu

Abstract

Kennell and Jamieson [KJ03] recently introduced the Genuinity system for authenticating trusted software on a remote machine without using trusted hardware. Genuinity relies on machine-specific computations, incorporating side effects that cannot be simulated quickly. The system is vulnerable to a novel attack, which we call a *substitution attack*. We implement a successful attack on Genuinity, and further argue this class of schemes are not only impractical but unlikely to succeed without trusted hardware.

1 Introduction

A long-standing problem in computer security is remote software authentication. The goal of this authentication is to ensure that the machine is running the correct version of uncorrupted software. In 2003, Kennell and Jamieson [KJ03] claimed to have found a software-only solution that depended on sending a challenge problem to a machine. Their approach requires the machine to compute a checksum based on memory and system values and to send back the checksum quickly. Kennell and Jamieson claimed that this approach would work well in practice, and they have written software called *Genuinity* that implements their ideas. Despite multiple requests Kennell and Jamieson declined to allow their software to be evaluated by us.

In this paper, we argue that

- Kennell and Jamieson fail to make their case because they do not properly consider powerful attacks that can be performed by unauthorized "imposter" software;
- Genuinity and Genuinity-like software is vulnerable to specific attacks (which we have implemented, simulated, and made public);
- Genuinity cannot easily be repaired and any software-only solution to software authentication faces numerous challenges, making success unlikely;
- proposed applications of Genuinity for Sun Network File System authentication and AOL Instant Messenger client authentication will not work; and
- even in best-case special purpose applications (such as networked "game boxes" like the Playstation 2 or the Xbox) the Genuinity approach fails.

To appreciate the impact of Kennell and Jamieson's claims, it is useful to remember the variety of approaches used in the past to authenticate trusted software. The idea dates back at least to the 1970s and led in one direction to the Orange Book model [DoD85] (and ultimately the Common Criteria Evaluation and Validation Scheme [NIS04]). In this approach, machines often run in physically secure environments to ensure an uncorrupted *trusted computing base*. In other contemporary directions, security engineers are exploring trusted hardware such as a secure coprocessor [SPWA99, YT95]. The Trusted Computing Group (formerly the Trusted Computing Platform Alliance) [Gro01] and Microsoft's "Palladium" Next Generation Security Computing Base [Mic] are now considering trusted hardware for commercial deployment. The idea is that trusted code runs on a secure processor that protects critical cryptographic keys and isolates security-critical operations. One motivating application is digital rights management systems. Such systems would allow an end user's computer to play digital content but not to copy it, for example. These efforts have attracted wide attention and controversy within the computer security community; whether or not they can work is debatable. Both Common Criteria and trusted hardware efforts require elaborate systems and physical protection of hardware. A common thread is that they are expensive and there is not yet a consensus in the computer security community that they can effectively ensure security.

If the claims of Kennell and Jamieson were true, this picture would radically change. The designers of Genuinity claim that an authority could verify that a particular trusted operating system kernel is running on a particular hardware architecture, without the use of trusted hardware or even any prior contact with the client. In their nomenclature, their system verifies the *genuinity* of a remote machine. They have implemented their ideas in a software package called *Genuinity*. In Kennell and Jamieson's model, a service provider, the *authority*, can establish the genuinity of a remote machine, the *entity*, and then the authority can safely provide services to that machine. Genuinity uses hardware specific side ef-

[*] This work was supported in part by DARPA, NSF, the US Postal Service, and Intel Corp. The opinions here are those of the authors and do not necessarily reflect the opinions of the funding sponsors.

fects to calculate the checksum. The entity computes a checksum over the trusted kernel, combining the data values of the code with architecture-specific *side effects* of the computation itself, such as the TLB miss count, cache tags, and performance counter values. Kennell and Jamieson restrict themselves to considering only uniprocessors with fixed, predetermined hardware characteristics, and further assume that users can not change hardware configurations. Unfortunately, as this paper demonstrates, even with Kennell and Jamieson's assumptions of fixed-configuration, single-processor machines, Genuinity is vulnerable to a relatively easily implemented attack.

To demonstrate our points, our paper present two classes of attacks—one class on the Genuinity implementation as presented in the original paper [KJ03], and more general attacks on the entire class of primitives proposed by Kennell and Jamieson. We wanted to illustrate these attacks against a working version of Genuinity, but Kennell and Jamieson declined to provide us with access to their source code, despite repeated queries. We therefore have attempted to simulate the main features of Genuinity as best we can based on the description in the original paper.

The designers of Genuinity consider two applications:

NFS: Sun's Network File System NFS is a well known distributed file system allowing entities (clients) to mount remote filesystems from an authority (an NFS file server). Unfortunately, NFSv3, the most widely deployed version, has no real user authentication protocol, allowing malicious users to impersonate other users. As a result, NFS ultimately depends on entities to run trusted software that authenticates the identities of the end users. Genuinity's designers propose using Genuinity as a system for allowing the authority to ensure that appropriate client software is running on each entity. The Genuinity test verifies a trusted kernel. However, a trusted kernel is not sufficient to prevent adversaries from attacking NFS: the weakness is in the protocol, not any particular implementation. We describe the NFS problem in more depth in Section 6.5.1.

AIM: AOL Instant Messenger AIM is a text messaging system that allows two entities (AIM clients) to communicate after authenticating to an authority (an AIM central server). AIM has faced challenges because engineers have reverse engineered AIM's protocol and have built unauthorized entities which the authority cannot distinguish from authorized entities. Kennell and Jamieson propose the use of Genuinity to authenticate that only approved client software is running on *entities*, thus preventing communication from unauthorized rogue AIM client software. As we discuss in Section 6.5.2 below, Genuinity will not work in these applications either.

In addition to these two applications, we consider a third application not discussed by Kennell and Jamieson:

Game box authentication Popular set-top game boxes such as Sony's Playstation 2 or Microsoft's Xbox are actually computers that support networking. They allow different users to play against each other. However, a widespread community of users attempts to subvert game box security (e.g., [Hua03]), potentially allowing cheating in online gaming. One might consider treating the game boxes as entities and the central servers as authorities and allowing Genuinity to authenticate the software running on the game boxes. This is arguably a best-case scenario for Genuinity: vendors manufacture game boxes in a very limited number of configurations and attempt to control all software configurations, giving a homogeneous set of configurations. However, even in this case, Genuinity fails, as we discuss in Section 7.2 below.

In short, we argue below that Genuinity fails to provide security guarantees, has unrealistic requirements, and high maintenance costs. More generally, our criticisms go to the heart of a wide spectrum of potential software-only approaches for providing authentication of trusted software in distributed systems. These criticisms have important consequences not only for Genuinity, but for a wide variety of applications from digital rights management to trusted operating system deployment.

Below, Section 2 summarizes the structure of Genuinity based on Kennell and Jamieson's original paper. Section 3 outlines specific attacks on Genuinity. Section 4 describes a specific *substitution attack* that can be used to successfully attack Genuinity and a specific implementation of that attack that we have executed. Section 5 details denial of service attacks against the current implementation of Genuinity. Section 6 describes a number of detailed problems with the Genuinity system and its proposed applications. Finally, Section 7 concludes by broadening our discussion to present general problems with software-only authentication of remote software.

2 A description of Genuinity

The Genuinity scheme has two parts: a checksum primitive, and a network key agreement protocol. The checksum primitive is designed so that no machine running a different kernel or different hardware than stated can compute a checksum as quickly as a legitimate entity can. The network protocol leverages the primitive into a key agreement that resists man-in-the-middle attacks.

Genuinity's security goal is that no machine can compute the same checksum as the entity in the allotted time without using the same software and hardware. If we substitute our data for the trusted data while computing the same checksum in the allowed time, we break the scheme.

As the authors of the original paper note, the checksum value can in principle be computed on any hardware platform by simulating the target hardware and software. The security of the scheme consequently rests on how fast the simulation can be performed: if there is a sufficient gap between the speed of the legitimate computation and a simulated one, then we can distinguish one from the other. Kennell and Jamieson incorporate side effects of the checksum computation itself into the checksum, including effects on the memory hierarchy. They claim that such effects are difficult to simulate efficiently. In Section 3, however, we present an attack that computes the correct checksum using malicious code quickly enough to fool the authority. A key trick is not to emulate all the hardware itself, but simply to emulate the effects of slightly different software.

Genuinity makes the following assumptions:

1. The entity is a single-processor machine. A multi-processor machine with a malicious processor could snoop the key after the key agreement protocol finishes.
2. The authority knows the hardware and software configuration of the entity. Since the checksum depends on the configuration, the authority must know the configuration to verify that the checksum is correct.
3. There is a lower bound on the processor speed that the authority can verify. For extremely slow processors, the claim that no simulator is fast enough is untrue.
4. The Genuinity test runs at boot time so the authority can specify the initial memory map to compute the checksum, and so the dynamic state of the kernel is entirely known.

Genuinity also makes the implicit assumption that all instructions used in computing the checksum are simulatable; otherwise, the authority could not simulate the test to verify that the checksum result is correct. As we discuss in Section 4.1.1, the precise-simulation requirement is quite stringent on newer processors.

In rest of this section we detail the Genuinity primitive, a checksum computation that the authority uses to verify the code and the hardware of the entity simultaneously. Following that, we review the higher level network key agreement protocol that uses the checksum primitive to verify an entity remotely.

2.1 The Genuinity checksum primitive

The checksum computation is the foundation of the Genuinity scheme. The goal of this primitive is that no machine with an untrusted kernel or different hardware than claimed will be able to produce a correct checksum quickly enough.

The details of the test are specified in the paper [KJ03] for a Pentium machine. First, the entity maps the kernel image into virtual memory using a mapping supplied by the authority, where each page of physical memory is mapped into multiple pages of virtual memory. This makes precomputation more difficult. Next, the authority sends a pseudorandom sequence of addresses in the form of a linear-feedback shift register. The entity then constructs the checksum by adding the one-byte data values at these virtual addresses. The original paper does not indicate how many iterations are performed during the course of the test. Between additions, the entity incorporates one of the following values into the checksum (the original paper under-specifies algorithmic details; see Table 2 for assumptions):

1. Whether a particular Instruction or Data TLB entry exists, and if so, its mapping. The original paper does not make clear which potential entries are queried (in addition, according to the Intel reference page [Int03], using the special test registers needed to access the TLB and cache data can lead to unpredictable results afterwards);
2. Instruction or data cache tags (again, the original paper does not indicate which cache entries to access);
3. A performance counter which measures the number of branch instructions encountered;
4. A performance counter which measures the number of instructions executed.

These processor-specific data sources are summarized in Table 1.

The authority must also compute the checksum. Since Kennell and Jamieson assume there is no fast simulator, the authority must compute the checksum natively or simulated it in advance. If the entity calculates the same checksum as the authority and returns it to the authority quickly enough, the entity is considered genuine.

The maximum allowable time is determined by the time it takes an "ideal simulator" to complete the task. An ideal simulator is one that receives the checksum code in advance, and can optimize the simulation code to run fast on the native hardware for that particular case. In the example given, Kennell and Jamieson report that the 133MHz Pentium legitimate entity took 7.93 seconds to complete the test, whereas the idealized simulator running on a 2.4GHz Pentium 4 took 10.72 seconds (35% longer than native execution); the latter figure was used

Table 1: **Processor-specific data included in the checksum.** In addition to checksumming over kernel image byte values, the checksum incorporates these pieces of processor specific data. The function used to incorporate the data is unspecified in the original paper, so our attack preserves references to this data exactly.

Data source	Description
Instruction TLB	ITLB miss count, physical page number
Data TLB	DTLB miss count, physical page number
Instruction cache	Cache line contents (which line is unspecified)
Data cache	Cache line contents (which line is unspecified)
Performance counter 1	Number of branches taken
Performance counter 2	Number of instructions executed

as the cutoff time for a response. The unaided (non-ideal) simulator required 18.53 seconds (134% longer). Our attack below (Section 3) inserts malicious code, computes the checksum correctly, and falls within the cutoff of 35% (Section 4.1.1).

Since the operations of integer addition and exclusive-or operate in different fields, the operation is nonlinear, and therefore *ex post facto* fixing up of the checksum is difficult. Any modification must be done on the fly, as the checksum is being computed.

2.2 The Genuinity key agreement protocol

The Genuinity checksum primitive is incorporated into a network key agreement protocol. At the end of a successful completion of the protocol, the authority will know that

1. The entity is running verified software on verified hardware, and
2. The entity is the one who passed the test if the key agreement succeeds.

The authority embeds its public key into the verified space of the Genuinity test to prevent man-in the middle attacks.

$E \rightarrow A$ The entity requests a challenge.

$A \rightarrow E$ The authority accepts the request, and sends the client a memory mapping to use during computation of the checksum. The virtual-to-physical page mappings are randomized, with many mappings pointing to the checksum code page. In particular, 2661 out of the 4096 total mappings pointed to the physical code page. The code contains many jumps to itself via alternate page mappings rather than local, relative jumps. These biases toward the code page are designed to make modification of the code more difficult.

$E \rightarrow A$ The entity notifies the authority of acceptance and installs the supplied memory mapping.

$A \rightarrow E$ The authority

1. sends the challenge (public key for the response and code for the checksum, both signed by the authority's key), and
2. starts the timer.

$E \rightarrow A$ The entity calculates the checksum using the initial memory map and the code that the authority sent. The entity encrypts the checksum and a nonce with the authority's public key and sends them to the authority.

$A \rightarrow E$ The authority stops the timer and checks if the checksum is correct. It sends either a qualification or rejection message to the entity.

$E \rightarrow A$ The entity uses periodic samples from the hardware cycle counter to generate as a symmetric session key. The entity encrypts the session key and a nonce with the authority's public key and sends them to the authority. The session key is never transmitted over the network.

3 Specific attacks against Genuinity

Attack overview We describe a specific attack on the Genuinity checksum primitive for the x86 architecture. We focus on x86 because it is the only one for which the algorithm is specified in in the original paper.

We were unable to obtain a copy of the code used in the original Genuinity paper. Therefore, our attacks refer to the published description of the algorithm; wherever we have had to make assumptions, we have documented them (see Table 2).

The premise of Genuinity is that if an entity passes the test, then that entity is running an approved operating system kernel on approved hardware. If we can insert a small amount of malicious code while still passing the test, then we can gain complete control of the system without being detected by the authority. In particular, once our modified checksum code succeeds, we have subverted the trusted exit path, which normally continues execution of the kernel. Instead, we may load any other kernel we wish, or send the session key to a third party.

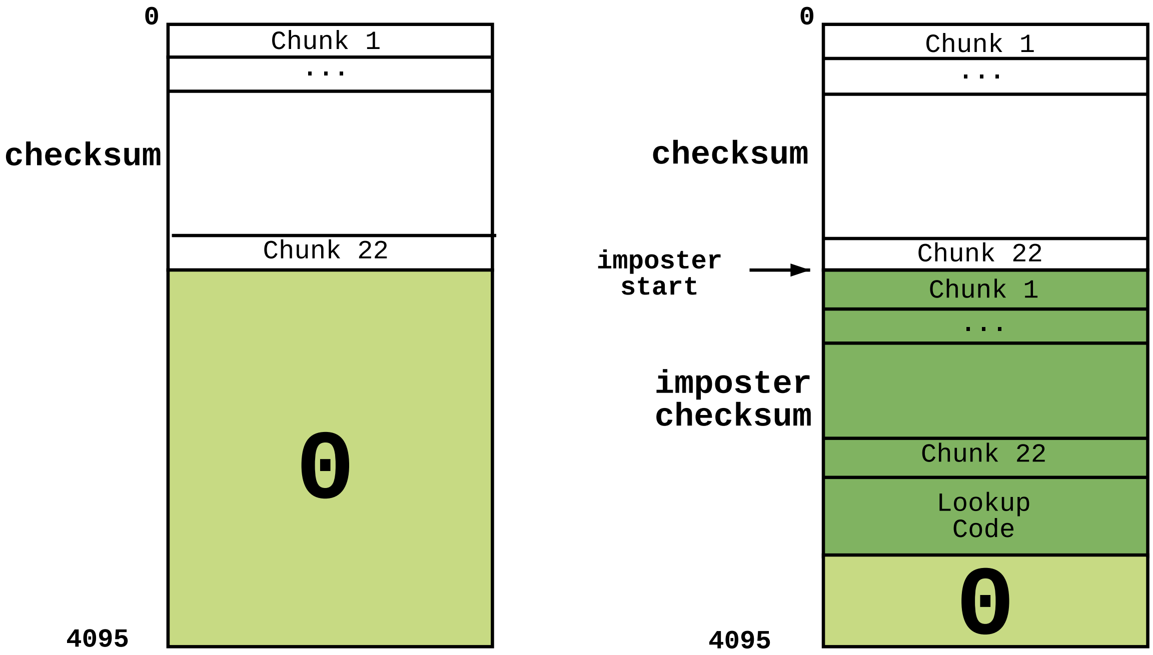

Figure 1: **The original checksum code page, and the malicious checksum code page.** The checksum code is divided into 22 code chunks. The imposter checksum code page replicates the original code entirely, then adds of imposter lookup code. The imposter lookup code checks each memory reference. If the address is in the imposter region (between `imposter start` and the end of the page), the lookup code returns 0 as the byte value. For all other memory references, the imposter lookup code returns the same value as the original lookup code.

4 Breaking Genuinity: substitution attacks

In this section, we describe two substitution attacks that work against the current implementation of Genuinity. The goal of a substitution attack is to modify the checksum code without modifying the checksum result. The first attack appends malicious code at the bottom of the checksum page. The second attack does not rely on extra space at the bottom of the checksum page.

4.1 The single page substitution attack

In the *single page substitution attack*, we append malicious checksum code on the same physical page as the original code; once it has computed the correct checksum, it can modify the machine's memory at will. Although the malicious code cannot initially be very large in order for the attack to work, we need only substitute enough to start loading arbitrary code.

This attack assumes there is extra space on the same page of physical memory as the checksum code page. We believe this is a reasonable assumption given Genuinity's description in the original paper; our own skeleton implementation of that algorithm consumed less than two kilobytes of memory of the four kilobyte page. Figure 1 illustrates the code page layout.

Table 1 describes processor-specific information that Genuinity uses to compute the checksum. The original paper does not specify how the data is incorporated into the checksum, but our attack is independent of the operation.

The checksum code is divided into 22 code *chunks*, called *nodes* in the original paper. 16 of these chunks, the memory chunks, choose a pseudorandom address in memory and add the value of the byte at that address to the checksum. The other 6 chunks incorporate the processor-specific data sources (TLB miss count, cache tags, etc.) described in Table 1. How the data is incorporated is not specified, so we preserve references to these sources exactly.

According to the original Genuinity paper, the checksum code fits into a single page in memory. Our implementation of the checksum code takes about 2KB, leaving about 2KB free with a page size of 4KB. The kernel used in the original Genuinity implementation is 2MB, but the virtual memory space over which the checksum is computed is 16MB since there are many-to-one mappings from virtual to physical pages. Since each page is 4KB, there are 4K virtual pages, but only 512 physical pages. Of the virtual pages, 65% (2661) of these map to the checksum code page, and about 3 virtual pages map to each of the other pages. For the sake of analysis, we assume the empty space on the checksum code page is

Table 2: **Assumptions made about the Genuinity algorithm in the absence of specification in the original paper.** Despite repeated requests of the original authors, we were unable to obtain a copy of the code for the Genuinity test. Our assumptions are used to fill in gaps in the published description.

Value/Behavior	Assumption
Layout of checksum code	Code at top of page, rest zero-filled
Iterations of compute chunks	16M iterations for each of 22 chunks
Operation to incorporate processor-specific data	exclusive-or
Frequency of incorporation of processor-specific data	6x for each iteration through 22 chunks
Which TLB entry or cache entry to examine	Unspecified (all values preserved in attack)

in the high memory of the page and is zero-filled. We discuss alternatives to this in Section 4.3. Figure 1 is our abstraction of the checksum code page.

The pseudocode representing the memory chunk is

```
memory_node:
  addr = next_LFSR()
  sum += memory[addr]
```

We rewrite the checksum code page maliciously. We can leave the original checksum code intact, and add imposter checksum code in the zero-filled space (Figure 1). To fool the authority into thinking we are running the original code, we need to know whether or not the pseudorandom address maps to the checksum code page. If the address does map to the checksum code page, the imposter checksum code must return the byte value from the original code page.

```
imposter_memory_node:
  addr = next_LFSR()
  if (is_codepage (addr) &&
      offset (addr) > imposter_start)
      // Preserve data cache, TLB
      temp = memory[addr]
      // Add the original byte value
      sum += 0
  else
      sum += memory[addr]
```

The difficulty lies in `is_codepage`: ideally a bit vector would represent the virtual memory space, where $\text{bit}_i = 1$ if virtual page i maps to the checksum code page. However, since we cannot make any memory references or use any variables without changing the data cache (Table 1), we must encode this vector in instructions. The bit vector requires 4K bits, or 128 32-bit words, to represent the page table. The vector is encoded by a sequence of 128 code chunks, one for each word. Each chunk loads one word of the vector into a register. We use the page number of the address to calculate the correct word of the bit vector, and jump to the corresponding code chunk. The chunk loads corresponding immediate word of the vector into a register, and we test the correct bit to see if the address is in the codepage.

```
is_codepage:
  // $r0 = virtual page number
  $r0 = addr >> 12
  // $r1 = bit index within the word
  $r1 = $r0 & 31
  // $r0 = which word to jump to
  $r0 = $r0 >> 5
  // Jump to the corresponding chunk
  jump ($r0*chunk_size) + chunk_base
chunk_base:
  // Chunk 1
  $r0 = immediate word1
  goto end
  // Chunk 2
  $r0 = immediate word2
  goto end
  ...
end:
   /* Test bit $r1 of $r0 */
   is_codepage = ($r0 & (1 << $r1))
```

Note that only two registers are used. Kennell and Jamieson designed the Genuinity algorithm not to access any data so as not to pollute the cache. It must therefore reserve two or three registers for temporary values in calculations. Our modifications do not need any additional registers for temporaries, and so are largely independent of the specifics of the Genuinity algorithm.

We have guaranteed that all memory reads will return the values for the original codepage—all that remains is to show that we can preserve the other invariants from Table 1.

1. Instruction TLB. Since the imposter checksum code resides on the same physical page as the original code, and we have not changed any page table entries, there are no changes to the ITLB. The miss count and contents are unaffected.
2. Data TLB. The imposter checksum code performs exactly the same memory loads as the original

code, so there are no changes to the DTLB.

3. Instruction cache. We preserve all cache entries. Cache lines corresponding to the original code never get loaded, so for accesses to them we substitute in the correct physical page number. This number is unambiguous, since there is only one instruction code page (containing both the imposter code and the original code).
4. Data cache. The imposter checksum code performs exactly the same memory loads as the original code, so there are no changes to the data cache.
5. Branch counter. On x86, there is an instruction to disable performance counters, including the branch counter. We can simply disable it before taking a branch that is not present in the original code, and re-enable it afterwards.
6. Instruction counter. As with the branch counter, it is possible to disable the instruction counter. Since we execute the same or more instructions per node, by disabling and re-enabling the counter at the right time, we can ensure that it holds the correct value for the original checksum code.

4.1.1 Prototype implementation

We implemented a rough prototype of our attack as a Linux loadable kernel module on a 1.5GHz Pentium 4 machine running the 2.4.20-28.9 Linux kernel. We did not have ready access to a Pentium machine, the processor used in the original paper. Using a kernel module allowed us to avoid rebooting and to disable interrupts as well as perform other privileged instructions needed to implement the Genuinity algorithm. Unfortunately, this approach made it impossible to remap the entire memory space; we performed our test in a reserved block of memory without remapping. Our implementation was in C with large blocks of inline assembly code to perform machine-specific instructions and to implement the `is_codepage` lookup function. Our implementation iterated 3000000 times over 16 memory reads interleaved with the processor-specific code chunks.

We learned a number of lessons in reproducing the Genuinity test. First, the special instructions used in the original test to access the instruction and data caches and the TLB directly are not supported on Intel processors after the Pentium. To the best of our knowledge, there are no available mechanisms to gain access to these structures in more recent Intel processors. In addition, the instruction counter did not return consistent results between trials. Intel does not guarantee the precision or reproducibility of performance counters; they are intended to be used as a guide for optimization. We therefore focused on two empirically repeatable counters that approximated those from the original Genuinity description: one that measured ITLB misses and one that measured the number of conditional branch instructions.

We successfully implemented our attack; we were able to compute the same checksum using the imposter code as when using the unmodified checksum code. The initial version of our attack code simply disabled the performance counters before running any added code, then re-enabled them before continuing. Unfortunately, the multipurpose instructions required to do this are serializing (preventing instruction-level parallelism) on the x86 and cause a significant slowdown; we stress that this is an artifact of the design of the instruction set architecture. On other architectures that provide dedicated instructions for this purpose, performance may be much better. In response, we modified our attack code to calculate the number of additional branches encountered and the number of additional ITLB misses generated by the attack and adjusted the counters appropriately.

The performance of the attack code, while not definitive in the absence of the original Genuinity code, was encouraging. We ran each test with and without inlining three times; the standard deviations in both cases were less than 0.6%.

With inlining turned off in the C compiler, the imposter code took 35% longer than the legitimate code (6.38s vs. 4.71s). This is the same as the 35% slowdown allowed by Genuinity as determined by the ideal simulator.

We also ran tests within inlining turned on. Due to suboptimal inlining by our C compiler, the best performance was obtained with no inlining. However, we found that inlining all but the bit vector lookup table of the imposter code could lead to a significant speedup. Inlining this portion in isolation gave an 18% speedup. Adding in the time to execute the lookup table yielded a net 42% slowdown over the fully inlined legitimate code. While this is not within the 35% boundary, in Section 4.2 we discuss using a higher clock speed machine to reduce the effective slowdown.

4.2 Improving attack performance

Suppose an adversary has an attack that computes the checksum while inserting malicious code, but the computation time does not fall inside the cutoff. The easiest way to improve the checksum computing performance is to increase clock speed. None of the side effects measures timing directly, because it is too difficult to get exactly repeatable results. Therefore, if all the CPU parameters except for clock speed are fixed, an adversary will compute the identical checksum value. This is easy to do, since typically CPUs in the same line are released at different clock speeds already. Another method would be to use a higher-performance main memory system, since main memory reads are the largest component of the overall time. This modification would not be reflected in the checksum value either. It is reasonable to expect that by claiming to have a 2 GHz Pentium 4 while actually having a 3 GHz machine—a 50% increase in clock speed—with an identical memory system, a considerable amount of additional code could be executed within the required time.

4.3 Countermeasures against substitution attacks

One can already see a kind of arms race developing: test writers might add new elements to the checksum, while adversaries develop additional circumventions. While it is possible to change the algorithm continually, it is likely that hardware constraints will limit the scope of the test in terms of available side effects; all an attacker must do is break the scheme on some hardware. While we believe that the attackers' ability to have the "last move" will always give them the advantage, we now consider some countermeasures and examine why they are unlikely to be significantly more difficult to accomodate than those we have already explored.

To prevent the single page substitution attack, Genuinity could fill the checksum code page with random bits.

Genuinity could also use different performance counter events or change the set used during the test. However, since the authority precomputes the checksum result, Genuinity must only use predictable counters in a completely deterministic way; we can compute the effects of our malicious code on such counters and fix them on the fly. For example, when the imposter checksum code starts executing instrutions that do not appear in the original code, it disables the instruction counters, and re-enables them after the extra instructions. Another possible solution which we did not implement is to calculate the difference in the number of instructions executed by the imposter code and the original code, and add this difference to the counter. We can treat other counters similarly.

At least two other improvements are suggested in the paper: self-modifying code and inspection of other internal CPU state related to instruction decoding. Since our attack code is a superset of the legitimate checksum code, and since we run on the same hardware (modulo clock speed) that we claim to have, neither of these seems insurmountable. Clearly, self-modifying code would require more sophisticated on-the-fly rewriting of the attack code, but by simply using a slightly faster machine (with the same TLB and cache parameters) this is easily overcome: the attack code is quite modular and easy to insert. As for inspection of instruction decoding, since the original code is a subset of our code, the internal state for the original instructions should be the same.

4.4 Response to countermeasures: the two page substitution attack

In Section 4.3, we describe some countermeasures Genuinity could take to prevent the single page substitution attack. We pick the first of these, filling the code page with random bits, and sketch a *two page substitution attack* that defeats this countermeasure.

Suppose Genuinity fills the unused code page with random bits, so the code page is not compressible. Then the single page substitution attack does not work and the imposter code must reside on a separate page.

We modify our attack somewhat to accomodate this change. The first step is to identify an easily-compressible page of code. Naturally, which particular page is most easily compressible will depend on the particular build. Simple inspection of a recent Linux kernel revealed that not only was the entire kernel compressible by a factor of 3 (the original `vmlinux` kernel vs. the compressed `vmlinuz` file), there were multiple 4K contiguous regions containing either all zeroes or almost all zeroes. Let us assume for the remainder of the discussion that the page is all zeroes; it would take only minor modifications to handle some non-zero values. In addition, since our hijacked page is referenced very infrequently (approximately one data read out of every thousand) that even if it took a little time to "uncompress" the data, this would likely not increase the execution time significantly.

The key step is to "hijack" the page and use it to store our imposter checksum code. The only memory region this step requires modifying is the hijacked page. This page, formerly zero-filled, now contains imposter checksum code.

The imposter code requires several fixups to preserve the invariants in Table 1.

The pseudocode looks like this:

```
imposter_memory_node:
  addr = next_LFSR()
  if page_number is hijacked_page
      // Preserve data cache
      temp = memory[addr]
      // Add the original byte value
      sum += 0
  else
      sum += memory[addr]
```

Let us review the checklist of invariants:

1. Instruction TLB. Instructions only come from only one physical page. To preserve references to the physical page number, we substitute the physical address of the original code page. To preserve the miss count, we can run the original checksum code

in advance and observe the TLB miss count whenever it is incorporated into the checksum. Eventually, this miss count should stabilize. Recall that the checksum code is divided into 22 code chunks, each of which refer to up to 2 virtual addresses. Since the instruction TLB on the Pentium is fully associative and contains 48 entries, all 44 of these virtual addresses fit into the ITLB. We estimate that the TLB should stabilize quickly, so the observation delay should not add significantly to the total time between receiving the challenge from the authority and sending our response. After observing the pattern of miss counts, the imposter checksum code can use these wherever the TLB miss count should be incorporated into the checksum.

In our implementation of the single page substitution attack, the ITLB miss count stabilizes after a single iteration through 22 code chunks, so this fixup is easy to accomplish.

2. Data TLB. The imposter checksum code performs exactly the same pattern of memory loads as the original code, so there are no changes to the DTLB.
3. Instruction cache. We simply fill the cache line with the contents of the original code page prior to executing the code to incorporate the cache data into the checksum. To do this, we need to encode the original checksum code in instructions, just as we did for the bit vector in the single page attack (Section 4.1). We unfortunately cannot read data directly from the original code page without altering the data cache.
4. Data cache. There is no change to the data cache, since the imposter code performs the same memory loads as the original code.
5. Branch counter, instruction counter. These are the same as in the original attack.

5 Breaking the key agreement protocol: denial of service attacks

At the key agreement protocol level, two denial of service attacks are possible. The first is an attack against the entity. Since there is no shared key between the authority and the entity (the entity only has the authority's public key), anyone could simply submit fake Genuinity test results for an entity, thereby causing the authority to reject that entity and force a retest. A retest is particularly painful, since the Genuinity test must be run on boot. Since the Genuinity test is designed to take as long as possible, this DoS attack requires minimal effort on the part of the attacker, since the attacker could wait as long as the amount of time a genuine entity would take to complete the test between sending DoS packets. It is possible that Genuinity could fix this problem by changing the key agreement protocol, but this attack works against the current implementation.

The second denial of service attack, analyzed in more depth in Section 6.2, is against the authority. Genuinity assumes that an adversary does not have a fast simulator for computing checksums, and so neither does the authority. The authority must precompute checksums, since the authority can compute them no more quickly than a legitimate entity. The original paper claims that the authority needs only enough checksums to satisfy the initial burst of requests. This is true only in the absence of malicious adversaries. It costs two messages for an adversary to request a challenge and checksum. The adversary can then throw away the challenge and repeat indefinitely. Further, the adversary can request a challenge for any type of processor the authority supports. The adversary can choose a platform for which the authority cannot compute the checksum natively. To make matters worse, the authority cannot reuse the challenges without compromising the security of the scheme, and might have to deny legitimate requests.

5.1 Countermeasures against DoS attacks

To avoid the denial of service attack against the client, Genuinity could assume that the client already has the public key of the authority.

The second denial of service attack is more difficult to prevent. The authority could rate limit the number of challenges it receives, but this solution does not scale for widely-deployed, frequently used clients such as AIM.

6 Practical problems with implementing the Genuinity test

We have presented a specific attack on the checksum primitive, and an attack at the network key agreement level. Genuinity could attempt to fix these attacks with countermeasures. However, even with countermeasures to prevent attacks on the primitive or protocol, Genuinity has myriad practical problems.

6.1 Difficulty of precisely simulating performance counters

Based on our experience in implementing Genuinity, we feel that it is likely to become increasingly difficult, if not impossible, to use many performance counters for a genuinity test. Not only are many performance counter values unrepeatable, even with interrupts disabled, they are the product of a very complex microarchitecture doing prefetching, branch prediction, and speculative execution. Any simulator—including the one used by the authority—would have to do a very low-level simulation in order to predict the values of performance counters with any certainty, and indeed many are not certain even on the real hardware! We do not believe that such simulators are likely to be available, let alone efficient, and may be virtually impossible; if the value of a performance counter is off by even one out of millions of samples, the results will be incorrect. This phenomenon is not surprising, since the purpose of the counters is to aid in debugging and optimization, where such small differences are not significant. The only counters that may be used for Genuinity are those that are coarser and perfectly repeatable: precisely the ones on which the effects of attack code may be easily computed in order to compensate for any difference. Finally, differences in counter architecture between processor families can seriously hamper the effectiveness of the test. Much of the strength of Genuinity in the original paper came from its invariants of cache and TLB information, much of which are no longer available for use.

6.2 Lack of asymmetry

Asymmetry is often a desirable trait in cryptographic primitives and other security mechanisms. We want decryption to be inexpensive, even if it costs more to encrypt. We want proof verification for proof-carrying code [Nec97] to be lightweight, even if generating proofs is difficult. Client puzzles [DS01] are used by servers to prevent denial of service attacks by leveraging asymmetry: clients must carry out a difficult computation that is easy for the server to check.

Genuinity, by design, is not asymmetric: it costs the authority as much, and likely more (because simulation is necessary), to compute the correct checksum for a test as it does for the client to compute it. This carries with it two problems. First, it exposes the authority to denial of service attacks, since the authority may be forced to perform a large amount of computation in response, ironically, to a short and easily-computed series of messages from a client. Second, it makes it no more expensive for a well-organized impostor to calculate correct checksums *en masse* than for legitimate clients or the authority itself. We shall explore this latter possibility further in Section 7.2.

6.3 Unsuitability for access control

The authors of the original paper propose to use Genuinity to implement certain types of access control. A common form of access control ensures that a certain user has certain access rights to a set of resources. Genuinity does not solve this problem: it does not have any provision for authenticating any particular user. At best, it can verify a client operating system and delegate the task to the client machine. However, we already have solutions to the user authentication problem that do not require a trusted client operating system: use a shared secret, typically a password, or use a public-key approach. Another kind of access control, used to maintain a proprietary interest, ensures that a particular application is being used to access a service. For example, a company may wish to ensure that only its client software, rather than an open-source clone, is being used on its instant-messenging network. In this case, the trusted kernel would presumably allow loading of the approved client software, but would also have to know which other applications *not* to load in order to prevent loading of a clone. The alternative is to restrict the set of programs that may be run to an allowed set, but it is unlikely that any one service vendor will get to choose this set for all its customers' machines.

6.4 Large Trusted Computing Base

When designing secure systems, we strive to keep the *trusted computing base* (TCB)—the portion of the system that must be kept secure—as small as possible. For example, protocols should be designed such that if one side cheats, the result is correct or the cheating detectable by the other side. Unfortunately, the entire client machine, including its operating system, must be trusted in order for Genuinity to protect a service provider that does not perform other authentication. If there is a local root exploit in the kernel that allows the user to gain root privilege, the user can recover the session key, impersonate another user, or otherwise access the service in an insecure way. Operating system kernels—and all setuid-root applications—are not likely to be bug-free in the near future. (A related discussion may be found in Section 6.5.1.)

6.5 Applications

Although two applications, NFS and instant messenging, are proposed by Kennell and Jamieson, we argue that neither would work well with the Genuinity test proposed, because of two main flaws: first, the cost of implementing the scheme is high in a heterogeneous environment, and second, the inconvenience to the user is too high in a widely distributed, intermittently-connected network.

6.5.1 NFS

The first example given in the original Genuinity paper is that an NFS server would like to serve only trusted clients. In the example, Alice the administrator wants to make sure that Mallory does not corrupt Bob's data by misconfiguring an NFS client. The true origin of the problem is the lack of authentication by the NFSv3 server itself; it relies entirely on each client's authentication, and transitively, on the reliability of the client kernels and configuration files. A good solution to this problem would fix the protocol, by using NFSv4, an NFS proxy, an authenticating file system, or a system like Kerberos. NFSv4, which has provisions for user authentication, obviates the need for Genuinity; the trusted clients merely served as reliable user authenticators.

Unfortunately, the Genuinity test does not really solve the problem. Why? The Genuinity test cannot distinguish two machines that are physically identical and run the same kernel. As any system administrator knows, there are myriad possible configurations and misconfigurations that have nothing to do with the kernel or processor. In this case, Mallory could either subvert Bob's NFS client or buy an identical machine, install the same kernel, and add himself as a user with Bob's user id. Since the user id is the only thing NFS uses to authenticate filesystem operations over the network once the partition has been mounted, Mallory can impersonate Bob completely. This requires a change to system configuration files (i.e., `/etc/passwd`), not the kernel. The bug is in the NFS protocol, not the kernel.

The Genuinity test is not designed to address the user-authentication problem. The Genuinity test does nothing to verify the identity of a user specifically, and the scope of its testing—verifying the operating system kernel—is not enough preclude malicious user behavior. Just because a machine is running a specific kernel on a specific processor does not mean its user will not misbehave. Further, even though the Genuinity test allows the entity to establish a session key with the authority, this key does no good unless applications actually use it. Even if rewriting applications were trivially easy (for example, IP applications could run transparently over IPSec), it does not make sense to go through so much work—running a Genuinity test at boot time and disallowing kernel and driver updates—for so little assurance about the identity of the entity.

6.5.2 AIM

The second example mentioned in the original Genuinity paper is that the AOL Instant Messenger service would like to serve only AIM clients, not clones. The Genuinity test requires the entity (AIM client) to be in constant contact with the authority. The interval of contact must be less than that required to, say, perform a suspend-to-disk operation in order to recover the session key. On a machine with a small amount of RAM, that interval might be on the order of seconds. On wide-area networks, interruptions in point-to-point service on this scale are not uncommon for a variety of reasons [LTWW93]. It does not seem plausible to ask a user to reboot her machine in order to use AIM after a temporary network glitch.

6.5.3 Set-top game boxes

Although the two applications discussed in the original paper are unlikely to be best served by Genuinity, a more plausible application is preventing cheating in multiplayer console games. In this scenario, Sony (maker of the Playstation) or Microsoft (maker of the Xbox) would use Genuinity to verify that the game software running on a client was authentic and not a version modified to allow cheating. This is a good scenario for the authority, since it needs to deal with only one type of hardware, specifically one that it designed. Even in the absence of our substitution attack (Section 4.1), Genuinity is vulnerable to larger scale proxy attacks (Section 7.2).

7 Genuinity-like schemes and attacks

We have described two types of attacks against this implementation of Genuinity: one type against the checksum primitive, and one type against the key agreement protocol. In this section we describe general attacks against any scheme like Genuinity, where

1. The authority has no prior information other than the hardware and software of the entity, and
2. The entity does not have tamper-proof or tamper-resistant hardware.

7.1 Key recovery using commonly used hardware

Clearly, the Genuinity primitive is not of much use if the negotiated session key is compromised after the test has completed. Since the key is not stored in special tamper-proof hardware, it is vulnerable to recovery by several methods. Many of these, which are cheap and practical, are noted by Kennell and Jamieson, but this does not mitigate the possibility of attack by those routes. Multiprocessor machines or any bus-mastering I/O card may be used to read the key off the system bus. This attack is significant because multiprocessor machines are cheap and easily available. Although the Genuinity primitive takes pains to keep the key on the processor, Intel x86 machines have a small number of nameable general-purpose registers and it is unlikely that one could be dedicated to the key. It is not clear where the key would be stored while executing user programs that did not avoid use of a reserved register. It is very inexpensive to design an I/O card that simply watches the system bus for the key to be transferred to main memory.

7.2 Proxy attacks: an economic argument

As we have seen, by design the authority has no particular computational advantage over a client or anyone else when it comes to computing correct checksums. Couple this with the fact that key recovery is easy in the presence of even slightly specialized hardware or multiprocessors, and it becomes clear that large-scale abuse is possible. Let us take the example of the game console service provider, which we may fairly say is a best case for Genuinity—the hardware and software are both controlled by the authority and users do not have as easy access to the hardware. In order to prevent cheating, the authority must ensure that only authorized binaries are executed. The authority must make a considerable investment in hardware to compute checksums from millions of users. However, this investment must cost sufficiently little that profit margins on a $50 or $60 game are not eroded; let us say conservatively that it costs no more than $0.50 per user per month. Now there is the opportunity for an adversary, say in a country without strict enforcement of cyberlaws, to set up a "cheating service." For $2 per month, a user can receive a CD with a cheat-enabled version of any game and a software update that, when a Genuinity test is invoked, redirects the messages to a special cheat server. The cheat server can either use specialized hardware to do fast emulation, or can run the software on the actual hardware with a small hack for key recovery. It then forwards back all the correct messages and, ultimately, the session key. The authority will be fooled, since network latency is explicitly considered to be unimportant on the time scale of the test.

7.3 A recent system: SWATT

More recently, the SWATT system [SPvDK04] of Seshadri et al. has attempted to perform software-only attestation on embedded devices with limited architectures by computing a checksum over the device's memory. Its purpose is to verify the memory contents of the device, not to establish a key for future use. Like Genuinity, SWATT relies on a hardware-specific checksum function, but also requires network isolation of the device being verified. As a result of restricting the domain (for example, the CPU performance and memory system performance must be precisely predictable), they are able to provide stronger security guarantees than Genuinity. SWATT requires that the device can only communicate with the verifier in order to prevent proxy attacks, which may hinder its applicability to general wireless devices. In addition, it is not clear that the dynamic state of a device (e.g., variable values such as sensor data or a phone's address book) can be verified usefully since an attacker might modify the contents of this memory and then remove the malicious code. Nevertheless, for wired devices with predictable state, SWATT provides a very high-probability guarantee of memory integrity at the time of attestation.

The authors of SWATT also present an attack on Genuinity. The attacker can flip the most significant bit of any bytes in memory and still compute the correct checksum with 50% probability.

8 Conclusion

Genuinity is a system for verifying hardware and software of a remote desktop client without trusted hardware. We presented an attack that breaks the Genuinity system using only software techniques. We could not obtain the original Genuinity code, so we made a best effort approximation of Genuinity in our attacks. Our substitution attacks and DoS attacks defeat Genuinity in its current form. Genuinity could deter the attacks with countermeasures, but this suggests an arms race. There is no reason to assume Genuinity can win it. Kennell and Jamieson have failed to demonstrate that their system is practical, even for the applications in the original paper. These criticisms are not specific to Genuinity but apply to any system that uses side effect information to authenticate software. Therefore, we strongly believe that trusted hardware is necessary for practical, secure remote client authentication.

Acknowledgements

We thank Rob Johnson for feedback and suggestions on the substitution attack. We also thank Naveen Sastry and David Wagner for many invaluable comments and insights. David Wagner also suggested the set-top game box application. Finally, we would like to thank the anonymous referees for several useful suggestions and corrections.

References

[DoD85] DoD. Standard department of defense trusted computer system evaluation criteria, December 1985.

[DS01] Drew Dean and Adam Stubblefield. Using client puzzles to protect TLS. In *10th USENIX Security Symposium*. USENIX Association, 2001.

[Gro01] Trusted Computing Group. Trusted computing group main specification, v1.1. Technical report, Trusted Computing Group, 2001.

[Hua03] Andrew Huang. *Hacking the Xbox: an introduction to reverse engineering*. No Starch Press, July 2003.

[Int03] Intel. Model specific registers and functions. http://www.intel.com/ design/intarch/techinfo/Pentium/mdelregs.htm, 2003.

[KJ03] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *12th USENIX Security Symposium*, pages 295–310. USENIX Association, 2003.

[LTWW93] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic.

In Deepinder P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[Mic] Microsoft. Next generation secure computing base. http://www.microsoft.com/resources.

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[NIS04] NIST. The common criteria and evaluation scheme. http://niap.nist.gov/cc-scheme/, 2004.

[SPvDK04] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.

[SPWA99] S. Smith, R. Perez, S. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*, October 1999.

[YT95] Bennett Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *First USENIX Workshop on Electronic Commerce*, pages 155–170, 1995.